Winter 1-1-1974

# Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs ; CU-CS-037-74

Leon J. Osterweil
*University of Colorado Boulder*

Lloyd D. Fosdick
*University of Colorado Boulder*

# Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs*

by
Leon Osterweil
and
Lloyd D. Fosdick
Department of Computer Science
University of Colorado
Boulder, Colorado 80302

Report #CU-CS-037-74 January 1974

Revised June 4, 1974

*This work supported by NSF Grant GJ-36461.

# Automated Input/Output Variable Classification as an Aid to Validation of FORTRAN Programs

by
Leon Osterweil
and
Lloyd D. Fosdick

ABSTRACT -- Certain types of errors in the coding of FORTRAN programs can be detected by careful analysis of the input/output usage of the variables in the program. It is easy to distinguish the value giving from the value receiving usages of variables within a statement. It is then easy to identify the input and output variables for statements and basic blocks. It is observed that a program variable must not be used as an input variable unless it has been used earlier in the program as an output variable. Conversely, once a variable has been used as an output variable, it should be expected that it will be used later in the program as an input variable. Algorithms are presented which employ depth-first searching techniques to verify whether input uses and output uses are improperly interspersed. These algorithms can also be used to determine the input and output parameters for entire subprograms. This capability extends the usefulness of these verification techniques, and can also be used in attempts to automate documentation production.

## Introduction

This paper presents some results which enable the automatic detection of certain semantically incorrect FORTRAN coding structures. We take the position that it is most difficult, perhaps impossible, to prove that a program is correct, but that it is often possible to detect errors, or at least highly questionable constructions, through more or less mechanical analysis techniques. One class of such techniques involves the study of the input/output behavior of a subprogram and its variables.

It is sometimes fruitful to think of a subprogram as being nothing more than a black box, which accepts certain values as inputs and produces others as outputs. The exact functional relationships between the inputs and outputs can then be said to completely describe the behavior of the subprogram. Clearly there are certain drawbacks to such an approach not the least of which is a definite loss of intuitive feel for what the subprogram is doing and hence whether it may for example be doing it inefficiently. We have found, however, that by adopting this view of subprogram behavior, it is possible to detect some semantic errors in programs.

In this paper we describe methods for identifying various classes of input and output variables within FORTRAN subprograms. We observe that a variable $\alpha$ which is found to be an input variable to a particular statement should have been an output variable from a previously executed statement. If there is no such previous statement, then the statement for which $\alpha$ is an input variable cannot be expected to execute properly, and an error in program logic is indicated. The use or potential use of an uninitialized variable, due perhaps to misspelling, can be detected in this way. Similarly if a variable $\beta$ is found to be an output variable from a particular statement, then we would ordinarily expect $\beta$ to be used as an input to some

subsequent statement. If $\beta$ is not, then a possible error is indicated. While this is not as sure a sign of erroneous code, it seems to us to be worth detecting.

Towards the detection of such errors and suspicious code sequences, we present algorithms to detect the variables which are input variables and output variables for entire subprograms. We first show that these algorithms are directly useful in the detection of some types of errors. We then show that certain slight modifications to the algorithms render them suitable for other kinds of error detection.

Definitions

Before proceeding further it is important to define some basic terms. We observe that virtually every reference to a variable in an executable FORTRAN statement refers to that variable in either a value giving or value receiving context. (It will be seen, however, that sometimes an external subprogram invocation statement may employ a variable reference in both giving and receiving contexts.) We say that a variable $\alpha$ appears in a value giving context in a statement provided that the statement cannot be executed successfully until and unless the value of $\alpha$ is obtained and used. For example, $\alpha$ is a value giver in each of the following FORTRAN statements:

```
X = A + α

ARRAY (α + 2) = 2.9

IF (α - 10) 50, 60, 70

DO 10 I = 1, α, 3
```

While the different statement types employ value givers in various ways, it is certainly possible to create a simple mechanical analysis routine capable of identifying the value givers for almost every statement type. Statements referencing external subprograms need special consideration and will be considered shortly.

Such a routine can certainly be created to identify variable usage in value receiving contexts as well. We say that a variable $\alpha$ is used in a value receiving context in a FORTRAN statement provided that as a result of the execution of that statement the value which $\alpha$ contained before execution of the statement (if any) is overwritten by a value generated during the execution of the statement. The most common example of a value receiving use of a variable is the use of a simple variable to the left of an equal sign in an assignment; e.g.

$\alpha = 0$

But $\alpha$ is also used as a value receiver in the following statements

DO 100 $\alpha$ = 1, 10

READ(5,25)$\alpha$

It is important to observe that a variable may be used both as a value giver and a value receiver within a single statement as in the following example:

$\alpha = \alpha + 1$

If a variable is used in a value giving context in a statement, we call it an input variable for that statement. Similarly a variable is called an output variable for a statement if it appears in that statement in a value receiving context.

In order to extend these concepts to entire subprograms more machinery is required. Let S be a FORTRAN subprogram consisting of the statements $s_1, s_2, \ldots, s_n$.* A basic block b, of S is a contiguous subset of the statements of S, $s_i, s_{i+1}, \ldots, s_{i+k}$, $k \geq 0$, having the property that no state-

*In the following discussion it is convenient to think of a logical IF statement as being not one, but rather two separate consecutively numbered statements, the first consisting of the letters IF and the parenthesized logical expression which follows, and the second consisting of the subsequent executable statement.

ment of b, except perhaps $s_i$ is the destination of any transfer of control statement anywhere in S. Hence we know that whenever $s_i$ is executed, $s_{i+1}$ will be executed next, then $s_{i+2}$,... then $s_{i+k}$. Hence it is reasonable to think of a basic block as a contiguous uninterruptible sequence of code which can be entered only at the top, and exited only at the bottom.

It is worthwhile to observe that we do not consider a statement which invokes an external subprogram to be a transfer of control statement, as it does not transfer control to another statement within its own subprogram. Hence, for example, a CALL statement need not terminate a basic block, but can be imbedded within one. In this respect, the definition of a basic block used here differs from some others in the literature.

Let $b_1$, ..., $b_n$ be the basic blocks of S. It is easy to see that the $b_j$ partition the statements of S into disjoint subsets.

It is now possible to form $G_S$, the program flow graph of S, in the following way. Let $B = \{b_1, \ldots, b_m\}$ be the vertex set of $G_S$, where it is assumed that $b_1$ is the basic block containing the unique entry point of S. Let the edge set of $G_S$ be the set of all ordered pairs of vertices $(b_x, b_y)$ having the property that it is possible to execute the first statement of $b_y$ immediately following the execution of the last statement of $b_x$. Hence the vertices of $G_S$ are the uninterruptible units of code, and the (directed) edges of $G_S$ are the possible transfers of control between units of code.

For the definitions of other graph theoretic terms used in this paper, see [1].

An execution sequence or control path for S is a sequence of basic blocks $b_{i_1}, b_{i_2}, \ldots, b_{i_x}$ for which $i_1 = 1$, $b_{i_x}$ is a basic block terminated by a STOP or RETURN statement, and for which $(b_{i_j}, b_{i_{j+1}})$ is an edge of $G_S$, $j = 1,2, \ldots, x-1$.

It is important to note that for a given subprogram, certain of its execution sequences may not actually be executable. For example in the following code sequence:

```
      IF(X.GT.0) GØ TØ 10
      IF(X.GT.0)I = I + 1
   10 . . .
```

it is not possible to ever execute the statement I = I + 1. Hence any execution sequence which includes this statement is in fact unexecutable. We call such sequences nonsemantic execution sequences. Execution sequences which are actually executable are called semantic execution sequences. It is clearly desirable to restrict attention to only the semantic execution sequences of a subprogram. Unfortunately the determination of whether an execution sequence is semantic or not seems to be extremely difficult. In fact, in general this determination is effectively undecidable. Hence, in what follows, we treat both semantic and nonsemantic execution sequences equally. As will be seen this weakens some of our results.

For a subprogram, S, suppose i is the number of a basic block in S and $\alpha$ is a variable used in S. Define:

$$I_{i,\alpha} = \{j \mid s_j \varepsilon b_i \text{ and } \alpha \text{ is an input variable to } s_j\}$$

$$O_{i,\alpha} = \{j \mid s_j \varepsilon b_i \text{ and } \alpha \text{ is an output variable for } s_j\}$$

Now define a minimum function as follows:

$$\text{Min } (X) = \begin{cases} \infty & \text{if } X = \emptyset \\ \min (X) & \text{if } X \neq \emptyset \end{cases}$$

where min is usual minimum function, having as its value the smallest value in the set X.

Now if $Min(I_{i,\alpha}) \neq \infty$ and $Min(I_{i,\alpha}) \leq Min(O_{i,\alpha})$ then $\alpha$ is defined to be an input variable to $b_i$. Otherwise it is a non-input variable to $b_i$.

If $Min(O_{i,\alpha}) \neq \infty$ then $\alpha$ is an output variable from $b_i$. Otherwise it is a non-output variable from $b_i$.

Now suppose $E = b_{i_1}, b_{i_2}, \ldots, b_{i_x}$ is an execution sequence for S. In a similar way define

$$I_{E,\alpha} = \{j \mid \alpha \text{ is an input variable to } b_{i_j}\}$$

$$O_{E,\alpha} = \{j \mid \alpha \text{ is an output variable from } b_{i_j}\}.$$

Now suppose $\alpha$ is a variable which either appears in a COMMON statement in S or in the formal parameter list for S. If there exists an execution sequence E for which

$Min(I_{E,\alpha}) \neq \infty$ and $Min(I_{E,\alpha}) \leq Min(O_{E,\alpha})$ then $\alpha$ is an input variable to S.

If no such E exists then $\alpha$ is a non-input variable to S. If for every execution sequence E, it is true that

$Min(I_{E,\alpha}) \neq \infty$ and $Min(I_{E,\alpha}) \leq Min(O_{E,\alpha})$

then $\alpha$ is called a strict input variable to S.

Similarly, suppose $\alpha$ is a variable which appears either in a COMMON statement in S or in the formal parameter list for S. If there exists no execution sequence, E, for which $Min(O_{E,\alpha}) \neq \infty$ then $\alpha$ is a <u>non-output variable</u> for S. If there exists such an E, then v is an <u>output variable</u> for S, and if for all execution sequences, E, $Min(O_{E,\alpha}) \neq \infty$, then $\alpha$ is a <u>strict output variable</u> for S.

Hence we recognize three classes of input variables--strict input, input, and non-input variables-- and three classes of output variables--strict output, output, and non-output. Thus, there are nine possible input/output classes for variables. Every COMMON variable and subprogram parameter falls into one of the nine classes.

We now present some algorithms for determining the input/output class of a given COMMON variable or subprogram parameter. As will be shown later, these algorithms can be rather easily adapted to the study of the input/output behavior of variables which are internal to individual subprograms.

The Identification of the Input-output Classes of COMMON and Parameter List Variables.

Before presenting the algorithms for identifying the input and output classes, it is important to discuss two obstacles to precise identification.

The first obstacle involves statements which invoke external subprograms. Earlier it was noted that it is straightforward to produce for virtually every statement type, mechanical analysis routines capable of identifying all value givers and all value receivers in any statement of the given type. CALL statements and function invocations are obvious exceptions to this claim. It is possible to determine the input and output parameters to an entire subprogram in the way which shall be demonstrated here, provided only that the subprogram invokes no other subprograms. (We shall refer to such subprograms as leaf subprograms.) Having made such analyses for leaf subprograms, however, the analyzer then knows the input and output variable behavior of all statements invoking such subprograms (e.g., CALL's). In this way, it is possible to determine the input and output classes of COMMON and parameter list variables for successively higher level subprogram units.

It must be acknowledged here that this bottom up analytic approach will fail for the case of a subprogram which can be called, perhaps indirectly, from a program which it can call itself. It is, of course, impossible to execute such a recursive sequence of procedure invocations in FORTRAN. However, it is possible to construct subprograms which contain such calling linkages, but

for which it is perhaps impossible to ever attempt the execution of the linkages which would force recursion. It is acknowledged that such constructions are possible. We feel, however, that such practices are unusual and represent poor coding practice. For these reasons we feel that the apparent failure of our algorithms to analyze such programs does not seriously diminish their usefulness.

The second obstacle involves array references. This is a more serious problem. It is possible and perhaps reasonable for different elements of a single array to exhibit different input/output behavior. For example, in SUBROUTINE S, below, A(1) and A(2) are clearly strict input, non-output variables, while A(3) and A(4) are clearly strict output non-input variables.

```
SUBROUTINE S(A)
DIMENSION A(4)
A(3) = A(1) + A(2)
A(4) = A(2) - A(1)
RETURN
END
```

This determination is easy to make in this case. If the subscripts used had been variables and expressions instead of constants, however, the analysis would have been far more difficult. Hence in our work we have made the simplifying decision that subscripted variables are to be treated no differently than simple variables. This is tantamount to the assumption that any reference to any element of an array in a particular context is equivalent to a reference to all elements of that array in that context. In addition, we have also assumed that whenever any variable is referenced in some context, all variables which are EQUIVALENCEd to it are also referenced in that same context. These assumptions can certainly cause us to make imprecise determinations of input-output behavior. In subroutine S, for example, we would conclude that A is a strict input, strict output parameter. We see no viable way to resolve this unfortunate weakness.

We now present inpvar, a procedure for determining the input class of v a given COMMON or calling sequence variable in a leaf subprogram, S, and outpvar, a procedure for determining the output class of v in S.

When inpvar terminates, it will be possible to use the variables strict and input in the following way to determine the status of v. If strict and input both are false, then v is a non-input variable. If input is true then v is an input variable.

If, in addition, strict is also true, then v is a strict input variable. The variables strict and output can be used analogously after outpvar has finished execution.

Program notes for procedure inpvar.

Global quantities.

procedures:

bboutpv (a, b) - Boolean procedure, with bboutpv assigned the value true if the variable b is an output variable for basic block a; otherwise it is assigned the value false.

bbinpv (a, b) - Boolean procedure, with bbinpv assigned the value true if the variable b, is an input variable for basic block a; otherwise it is assigned the value false.

arrays:

outdegree [a] - The outdegree of the vertex representing basic block a in the graph of the subprogram.

head [a, b] - The basic block at the head of the edge b from the basic block a.

variables:

n - The number of basic blocks, an integer.

Numbering conventions.

Basic blocks of a subprogram are assumed to be numbered 1, 2, ..., n where n is the number of basic blocks in the subprogram.

The unique entry vertex of the subprogram graph represents basic block 1.

The edges from a vertex are numbered 1, 2, ..., $n_e$ where $n_e$ is the outdegree of the vertex.

```
procedure subprogram inpvar (v, input, strict);
   integer v; Boolean input, strict;
   begin
      Boolean array visited [1: n]; integer j;
      comment basic blocks are numbered 1, 2, ..., n;
      procedure inpvar (basic block);
         integer basic block;
      begin
         integer edge;
         visited [basic block] := true;
         if bbinpv (basic block, v) then input := true
         else
            if (outdegree [basic block] = 0 ∨ bboutpv (basic block, v)) then
               strict := false
            else
            for edge := 1, edge + 1 while
               ((edge ≤ outdegree [basic block]) ∧ (strict ∨ ¬input))
            do
            begin
               if ¬ visited [head [basic block, edge]] then
                     inpvar (head [basic block, edge]);

            end
      end inpvar;
```

```
        strict := true; input := false;
        for j := 1 step 1 until n do visited [j] := false;
        inpvar (1)
    end subprogram inpvar
```

Program notes for procedure outpvar.

Global quantities.

procedures:

bboutpv (a, b) - Boolean procedure, with bboutpv assigned the value true if the variable b is an output variable for basic block a; otherwise it is assigned the value false.

arrays:

indegree [a] - The indegree of basic block a in the subprogram graph.

tail [a, b] - The basic block at the tail of the edge b to basic block a.

leaf [j] - The basic block which is the $j^{th}$ leaf (i.e. a vertex with with outdegree = 0) of the subprogram graph.

variables:

n - The number of basic blocks in the subprogram graph.

number of leaves - The number of leaves in the subprogram graph.

```
    procedure subprogram outpvar (v, output, strict)
        integer v; Boolean output, strict;
        begin
            Boolean array visited [1: n]; integer j;
            comment basic blocks are numbered 1, 2, ..., n;
            procedure outpvar (basic block);
                integer basic block;
            begin
```

```
integer edge;
visited [basic block] := true
if bboutpv (basic block, v) then output := true
else
   if indegree [basic block] = 0 then strict := false
   else
   for edge := 1, edge + 1 while
      ((edge ≤ indegree [basic block] ∧ (strict ∨ ¬ output))

   do
   begin
      if ¬visited [tail[basic block, edge]] then
           outpvar (tail [basic block, edge]);

   end
end outpvar;
strict := true; output := false;
for j := 1 step 1 until n do visited [j] := false;
for j := 1 step 1 until number of leaves do
 outpvar (leaf[j])
end subprogram outpvar
```

It is important to observe that outpvar may not yield valid results if there are vertices of $G_S$ which cannot be reached from the start vertex. This does not appear to be a harsh assumption as it eliminates only subprograms having segments of code which can never be executed. Such routines have an obvious flaw which should be fixed before attempting the more complex analysis discussed here.

Algorithms inpvar and outpvar are efficient. Both require that each edge of $G_S$ be traversed at most once. Hence the running time of each is at most proportional to the number of edges in $G_S$. Moreover, by examining the storage requirements of the various auxiliary arrays, it becomes clear that the storage required to execute these algorithms is at most proportional to the sum of the number of vertices and edges of $G_S$.

Both algorithms can be readily adapted to perform their tasks in parallel for several variables at once. There is no reason why we must determine input or output status for individual variables one at a time. Instead it is possible and preferable to execute each algorithm only once, for the set of all variables appearing either in COMMON or as bound variables.

As already noted inpvar and oupvar work properly for leaf subprograms. These subprograms have the property that every input variable to a block is a strict input variable, and every output variable from a block is a strict output variable. Since non-leaf subprograms do not in general have this property, some modifications must be made to the algorithms for such programs. Specifically, inpvar can be used to determine the input status of v in a non-leaf subprogram if the algorithm is augmented so that strict is set equal to false every time a block using v as a nonstrict input variable is encountered during execution. A similar minor modification to outpvar will likewise enable it to work correctly on non-leaf subprograms. Of course, the modified outpvar and inpvar continue to work correctly on leaf subprograms.

## Applications of Input and Output Class Determination for Variables

Now that it has been demonstrated that the input and output classes of COMMON and parameter list variables to FORTRAN subprograms can be efficiently determined automatically, several applications become evident. Probably the most obvious of these is as an automated documentation tool. By using the methods outlined above, it is easy to automatically obtain the identities of all variables which carry values into a given subprogram, as well as those which carry values out. This is perhaps more useful as a check on existing documentation than as a way of generating documentation from scratch. It is quite conceivable that erroneous coding might be detected more readily if the input-output status of subprogram variables is determined to be different from the status asserted by the programmer in his documentation. Such unexpected results can be assumed to be stimuli to sharply focused debugging efforts. This application is philosophically akin to a number of efforts (see, for example, [2], [3], [4], [5], and [6]) in the general area of verifying assertions about programs.

Another obvious use is in verifying that values are not passed back to calling subprograms through non-variables. Suppose S is a subprogram for which the input-output status of its parameters is known. If S is invoked with constants and/or expressions used as arguments, then it is important that these non-variables be associated with formal parameters which are non-output parameters from S. Anything else would cause a potential violation of the ANSI standard. It is now possible to make such determinations.

It is also worth observing that non-input, non-output parameters serve no purpose in the formal parameter list of a subprogram. While it is unlikely that putting such a variable in the parameter list of a subprogram will cause an error, it is conceivable that it is symptomatic of some other error (e.g. a misspelled name), however, or at best represents a coding inefficiency. For these reasons, it seems reasonable to alert the programmer to the existence of such parameters.

It is far more interesting, however, to reflect on the ways in which these techniques can be employed to automatically detect semantic irregularities in FORTRAN programs. Suppose that subprogram S is a leaf subprogram, and that the input-output behavior of its variables has been analyzed. Assume that v, a formal parameter of S, is found to be a strict input parameter to S. Suppose T is a subprogram which invokes S, passing a value to the parameter v through the argument variable w. Because v is a strict input parameter to S, we cannot expect S to function properly within the ANSI FORTRAN standard unless w receives a value prior to any invocation of S from T. Thus suppose E is an execution path for T in which S is invoked during the execution of block $b_{i_j}$. S cannot be expected to execute correctly unless w is an output variable from some block, $b_{i_k}$, where $k < j$; where it is assumed here for simplicity that w is not an output variable in any statement of $b_{i_j}$ prior to the invocation of S. If there exists an execution path which does not result in w being an output variable before the invocation of S, then we must assume that the path is never actually executed, or that its execution will cause an error. In either case, the programmer should be advised of a probable error.

The above situation is readily detected with only the algorithmic tools already at hand. Let us suppose that S is invoked during block $b_{i_j}$ of T and that w is not used as an output from any statement prior to the invocation within $b_{i_j}$, as stated above. We consider $b_{i_j}$ to be a stop vertex of T, and use procedure outpvar to determine the output status of the variable w. If w turns out to be strict output, no trouble is suspected. If not, there are at least some paths which, if executed, will cause errors. If w turns out to be non-output a serious coding error is indicated, for no matter how $b_{i_j}$ is reached, an error must result when S is invoked.

In the case where v is simply an input variable (but not a strict input) to S we can no longer be certain of such errors. It is perhaps worthwhile to print out a warning that all (or even some) execution sequences do not initialize w before invocation of S, but this can at best be a tentative warning with slight conviction behind it. Here we see the weakening effects of not being able to distinguish between the semantic and nonsemantic execution sequences of a subprogram. Perhaps v is used first as an input parameter for all semantic execution sequences of S, but not for some non-semantic execution sequences. In such a case v is semantically a strict input variable. Hence even if all execution sequences of T (even all semantic execution sequences of T) fail to initialize w before invoking S, however, we cannot be sure we have detected an error.

There is an analog to the above situation in which errors are identified with more difficulty and less certainty. Let us suppose now that v is a non-input, strict output parameter from S, whose value is picked up by w in block $b_{i_j}$ of subprogram T. We would normally expect that w would be an input variable to some block of T executed subsequent to $b_{i_j}$ or some statement of $b_{i_j}$ executed subsequent to the invocation of S. It is easy to check for this situation by

using procedure inpvar in a way analogous to the way in which procedure outpvar is used above. If w is not used as input for any execution path subsequent to $b_{i_j}$, it is probably worthwhile to alert the programmer to this situation. Unfortunately this situation is not certain to be symptomatic of an error. It is possible that the value placed into v by S is needed only after some invocations of S, but not after all. If the value placed into v by S is never used as input in any block following any invocation of S in any subprogram which ever invokes S, then it is probably reasonable to suspect a coding error, or at least wasted effort within subprogram S. Unfortunately the detection of this error is more complex and time consuming than the detection of the previous case.

It is now clear that inpvar and outpvar are valuable in identifying inconsistencies between the usage of variables in calling and in called subprograms. This is done essentially by first studying a called subprogram to determine which calling sequence variables are input and which are output, and then by studying the contexts of all statements involving invocations of the subprogram to determine whether input variables have been previously assigned values, and whether the values of output variables are subsequently used.

It is clear that these techniques are perfectly applicable to all executable statements, not just those invoking external subprograms. It is certainly easier to make input and output variable determinations for statements not involving external routines than for those which do. Once those determinations are made for a given statement it is possible to use the exact techniques described above to study the context of that statement. As before we are interested in whether input variables have been previously assigned values and whether the values in output variables are subsequently used.

There seems to us to be a slight difference in the severity of output variable misusages, however. We feel that variables which are outputs from

non-external subprogram invocation statements should more strongly be expected to be used subsequently as inputs. With the exception of the non-utilization of a DO-index or subprogram parameter, there seems to be little excuse for the creation of an output value which is never to be used as an input. Hence for instance if the final use of a variable is as an output from a non-external subprogram invocation statement which is not a DO, then a serious error is indicated unless the variable is in COMMON or the parameter list.

The previous discussion suggests that inpvar and outpvar could and should be modified to perform useful checking on variables which are internal to a given subprogram. For each such internal variable, v, a routine much like inpvar could be used to verify that v is first used as an output variable. If there is any path for which this is not true, an error should be indicated. Similarly a routine much like outpvar could be used to verify that the last use of v is as an input variable, or perhaps as an output from a DO or external subprogram invocation statement. If there is any path for which this is not true, an error is likely.

## Conclusions

We feel that this work is a first step towards the goal of computer aided verification of FORTRAN programs. As noted earlier, we have not addressed ourselves to the task of proving correctness of programs, but rather have confined ourselves to searching for certain or highly likely errors. We have demonstrated that it is possible to automatically detect certain structural and semantic deficiencies. It is expected that packages utilizing these techniques will prove useful to programmers by helping them to find errors or weaknesses in their programs. Our goal, however, is to produce comprehensive packages of such tools to be placed in the hands of independent evaluators. The purpose of such packages would be to increase the certainty of the evaluator that a given FORTRAN program is reliable and correct.

References

1. Frank Harary, Graph Theory, Addison Wesley, 1969.

2. J. King, A Verifying Compiler, Debugging Techniques in Large Systems (R. Rustin, ed.), Prentice Hall, 1971, pp. 17-39.

3. Hetzel, W. C. (Ed) Program Test Methods, Prentice Hall, 1973.

4. Floyd, R. W. Assigning meanings to programs. In Proceedings of Symposia in Applied Mathematics, J. T. Schwartz (Ed), Vol. 19, 1966, pp. 19-32.

5. London, Ralph L. Bibliography on proving the correctness of computer programs. In Machine Intelligence 5, 1970, pp. 469-580.

6. London, Ralph L. U. Wisconsin Computer Science Dept., Report No. 104 (Dec. 1970), 1-8.